LocFaults: A new flow-driven and constraint-based error localization approach

Mohammed Bekkouche, Hélène Collavizza, Michel Rueher

LocFaults: A new flow-driven and constraint-based error localization approach*

Mohammed Bekkouche
University of Nice–Sophia
Antipolis, I3S/CNRS
BP 121, 06903 Sophia
Antipolis Cedex, France
Mohammed.Bekkouche@i3s.unice.fr

Hélène Collavizza
University of Nice–Sophia
Antipolis, I3S/CNRS
BP 121, 06903 Sophia
Antipolis Cedex, France
helen@polytech.unice.fr

Michel Rueher
University of Nice–Sophia
Antipolis, I3S/CNRS
CS 40121, 06903 Sophia
Antipolis Cedex, France
Michel.Rueher@unice.fr

ABSTRACT

We introduce in this paper LocFaults, a new flow-driven and constraint-based approach for error localization. The input is a faulty program for which a counter-example and a postcondition are provided. To identify helpful information for error location, we generate a constraint system for the paths of the control flow graph for which at most k conditional statements may be erroneous. Then, we calculate Minimal Correction Sets (MCS) of bounded size for each of these paths. The removal of one of these sets of constraints yields a maximal satisfiable subset, in other words, a maximal subset of constraints satisfying the post condition. To compute the MCS, we extend the algorithm proposed by Liffiton and Sakallah [21] in order to handle programs with numerical statements more efficiently. The main advantage of this flow-driven approach is that the computed sets of suspicious instructions are small, each of them being associated with an identified path. Moreover, the constraint-programming based framework of LocFaults allows mixing Boolean and numerical constraints in an efficient and straightforward way. Preliminary experiments are quite encouraging.

*This work was partially supported by ANR VACSIM (ANR-11-INSE-0004), ANR AEOLUS (ANR-10-SEGI-0013), and OSEO ISI PAJERO projects.
Part of this work was done while Michel Rueher was visiting professor at NII (National Institute of Informatics), Tokyo.

SAC'15 April 13-17, 2015, Salamanca, Spain.

http://dx.doi.org/10.1145/2695664.2695822

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Constraints; D.2.5 [**Testing and Debugging**]: Debugging aids, Diagnostics, Error handling and recovery

General Terms

Verification, Algorithms

1. INTRODUCTION

Error localization from counter-examples and associated execution traces is a crucial issue in the software development process. Indeed, when a program P contains errors, a model checker usually provides a counter-example and an execution trace that is too long and too difficult to understand. This kind of outputs is therefore of limited interest for the programmer who has to debug the program. Thus, identifying code portions that may contain errors is often a difficult and expensive process, even for experienced programmers.

That is why we introduce here a new flow-driven and constraint-based approach for error localization. This new approach takes advantage from the structure of the Control Flow Graph (CFG) as well as from the flexibility provided by the constraint-programming framework. The process starts with a faulty program and a counter-example violating the postcondition. To provide helpful information for finding potential errors, we generate a constraint system for the paths of the CFG for which at most k conditional statements may be erroneous. Then, we calculate Minimal Correction Sets (MCS) of bounded size for each of these paths. In other words, we bound both the number of suspected assignments on the initial path, and the number of deviations from that path. To compute MCS, we extend the algorithms introduced by Liffiton and Sakallah [21, 20] to be able to handle programs with numerical statements more efficiently. Note that LocFaults may miss some errors since the number of deviations as well as the size of the MCS are bounded to limit combinatorial explosion.

To sum up, we take advantage of the information of the CFG for computing small sets of suspicious instructions, each of them being associated to an identified path. Moreover, this constraint-programming based framework provides an efficient and straightforward way for mixing Boolean and numerical constraints.

The rest of the paper is organized as follows. Section 2 illustrates how LocFaults works on a small example. Sec-

tion 3 goes into detail of the LocFaults framework. Section 4 reports experimental results on a number of benchmarks and problems, comparing our approach with BugAssist, a state-of-the art error localization framework [17, 18]. Section 5 discusses related work, summarizes the contributions and presents future research directions.

2. MOTIVATING EXAMPLE

Consider program `AbsMinus` (see fig. 1). The inputs are integers $\{i,\ j\}$ and the expected output is the absolute value of $i - j$. An error has been introduced in line 10, thus for the input data $\{i = 0, j = 1\}$, program `AbsMinus` returns -1. The postcondition here is just $result = |i - j|$[1].

```
1 class AbsMinus {
2 /*returns |i−j|,the absolute value of i minus j*/
3 /*@ ensures
4  @ ((i < j) ==> (\result == j−i)) &&
5  @ ((i >= j) ==> (\result == i−j)); */
6  int AbsMinus (int i, int j) {
7    int result;
8    int k = 0;
9    if (i <= j) {
10       k = k+2; } // error : k = k+2 instead of k=k+1
11   if (k == 1 && i != j) {
12       result = j-i; }
13    else {
14       result = i-j; }
15 return result;
  }
}
```

Figure 1: Program AbsMinus

The CFG of program `AbsMinus` and a faulty path are depicted in figure 2. This faulty path corresponds to the input data : $\{i = 0, j = 1\}$. First, LocFaults collects on path 2.(b) the constraint set $C_1 = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = k_0 + 2, r_1 = i_0 - j_0\}$[2]. Then, LocFaults computes the MCS of C_1. It is important to stress that the constraints defining the assignments of the input variables cannot belong to the computed MCS. Indeed, relaxing these constraints, rather corresponds to the generation of some test cases that satisfy the postcondition. So, only one MCS can be found in C_1: $\{r_1 = i_0 - j_0\}$. In other words, if we assume that the conditional statements are correct, the only suspicious statement on this faulty path is statement 14.

Then, LocFaults starts the deviation process. The first deviation (see figure 3.(a), green path) still produces a path that violates the postcondition. Thus, it is rejected. The second deviation (see figure 3.(b), blue path) produces a path that satisfies the postcondition. So, LocFaults collects the constraints on the part of path 3.(b) which precedes the deviated condition, that is $C_2 = \{i = 0, j = 1, k_0 = 0, k_1 = k_0 + 2\}$. Then LocFaults searches for an MCS of $C_2 \cup \neg(k = 1 \wedge i \neq j)$.That is to say, one tries to identify the assignments which must be modified to force the program to follow a path that satisfies the postcondition. Therefore, for this second deviation two suspicious statements are identified:

- The conditional statement in line 11;
- The assignment in line 10 since the corresponding constraint is the only MCS of $C_2 \cup \neg(k = 1 \wedge i \neq j)$.

Then, LocFaults goes on and tries to deviate a second condition. The only possible path is the one where both conditions of program `AbsMinus` are deviated. However, since it has the same prefix than the first deviated path, we discard it.

This example shows that LocFaults produces relevant and helpful information on each faulty path. Unlike to BugAssist, a state of art system, it does not merge all suspicious statements in a single set, which may be difficult to exploit by the user.

3. THE LocFaults FRAMEWORK

In this section we first introduce the formal definitions of MUS, MSS and MCS which are the basis of our error-localization framework. Then, we give an overview of BMC (Bounded Model Checking) based on constraint programming. Finally, we detail how we compute some MCS of bounded size along the program paths, using a depth-first search on the CFG of the program.

3.1 Error localization and MCS computation

Since we encode the set of statements of a faulty path as a set of constraints, the error localization problem is similar to the problem of finding a correction set for an inconsistent constraint system. This problem has been addressed both in the operational research community and constraint community. When searching useful information to correct inconsistent constraint systems, one may look for two kinds of information:

1. How many constraints in an unsatisfiable set of constraints can be satisfied ?
2. Which part of the constraint system is unsatisfiable ?

To answer these questions, the notion of MUS, MSS and MCS have been introduced by Liffiton and al [21].
A Minimal Unsatisfiable Subsets of constraints (MUS), also called "unsatisfiable cores" is an unsatisfiable system of constraints such that removing any one of its elements makes the remaining set of constraints satisfiable:

$$M \subseteq C \text{ is a MUS} \Leftrightarrow M \text{ is UNSAT}$$
$$\text{and } \forall c \in M : M \setminus \{c\} \text{ is SAT.}$$

A Maximal Satisfiable Subset (MSS) is a generalization of MaxSAT and MaxCSP where we consider the maximality instead of the maximum cardinality

$$M \subseteq C \text{ is an MSS} \Leftrightarrow M \text{ is SAT}$$
$$\text{and } \forall c \in C \setminus M : M \cup \{c\} \text{ is UNSAT.}$$

The definition of the MSS is very close to that of IIS (Irreducible Inconsistent Subsystem) used in operational research [5, 6, 7].
An MCS is a subset of the constraints of an infeasible constraint system whose removal yields a satisfiable set of constraints ("correcting" the infeasibility). Furthermore it is minimal in the sense that any proper subset does not satisfy this property [21].

$$M \subseteq C \text{ is an MCS} \Leftrightarrow C \setminus M \text{ is SAT}$$

[1]Specifications are written in JML `http://www.eecs.ucf.edu/~leavens/JML/index.shtml`

[2]Before collecting the constraints, a variable renaming is required. More precisely, the program has to be transformed in DSA form [2] which ensures that every variable is assigned at most once along any path.

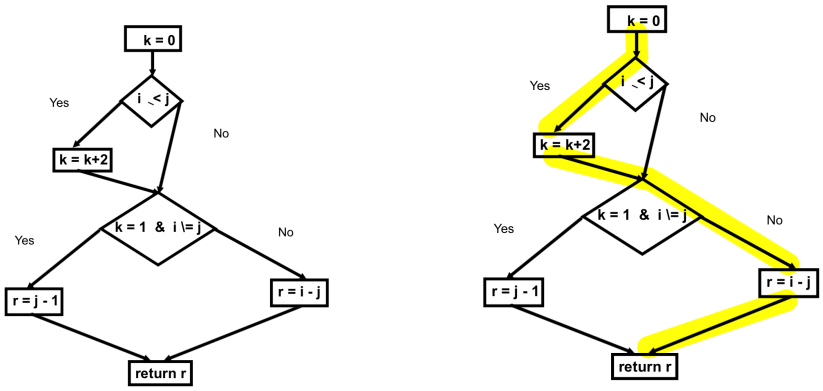

Figure 2: CFG and faulty path – Program `AbsMinus`

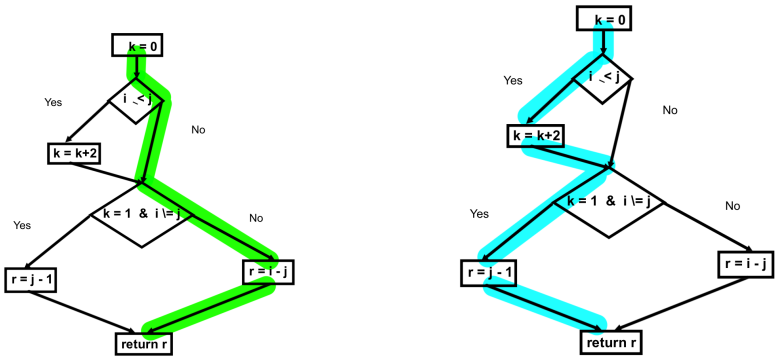

Figure 3: Paths with one deviation – Program `AbsMinus`

and $\forall c \in M : (C \setminus M) \cup \{c\}$ is UNSAT.

There is a duality relationship between the set of MUS and the set of MCS [4, 21]

Different algorithms have been proposed for calculating IIS/MUS et MCS, e.g. `Deletion Filter`, `Additive Method`, `Additive Deletion Method`, `Elastic Filter` [5, 26, 6, 7]. Junker [19] proposed an algorithm based on a generic "Divide-and-Conquer" strategy to efficiently compute IIS or MUS when the cardinality of the conflict-subsets is much smaller than the one of the whole constraint set.

The algorithm introduced by Liffiton and Sakallah [21] first calculates all MCS in an increasing order, then all MUS by using the above property mentioned. The algorithm presented in section 3.4 is derived from that algorithm.

Various improvements of these algorithms [10, 20, 23] have been proposed during the last years but they are closely related to the specificity of SAT solvers, and thus, they are quite difficult to transpose to numeric constraint solvers.

Next section introduces the BMC framework we use to collect the constraints on a faulty or suspicious path.

3.2 Constraint based BMC

Constraint based BMC is a BMC approach that uses constraints for modeling the program and its specification, and constraint solving for checking if the program conforms its specification [8, 9]. BMC uses a bound b to unfold loops: they are replaced with conditional statements of depth b. If an error is found for a given b, then the program does not conform its specification. Otherwise, b is increased for searching a deeper error, until a maximum value of b has been reached.

Let $PROG_b$ be the program after being unfolded b times. One BMC step checks the Hoare triplet $\{PRE, PROG_b, POST\}$ where PRE is the precondition and $POST$ is the postcondition. $PROG_b$ does not conform its specification if the formula $\Phi = PRE \wedge PROG_b \wedge \neg POST$ is satisfiable. An instantiation of the variables of Φ is then a *counter-example* because it satisfies both the precondition and the program, but it does not satisfy the postcondition.

CPBPV [8] is a BMC tool based on constraint programming. $CPBPV$ translates PRE and $POST$ into constraints, and transforms $PROG_b$ into a CFG whose nodes are the conditions and assignments of the program translated into constraints[3]. CPBPV builds the constraint system CSP associated to formula Φ *on the fly*, using a depth-first search on the data structure of the graph. At the initial state, CSP contains the constraints based from PRE and $\neg POST$. Then the constraints of a path are added during the graph exploration. When the last node has been reached on a path, the satisfiability of CSP is checked. When CSP has a solution, an error has been found in the program thus the BMC process is stopped. Otherwise, another branch is explored. When all the branches have been explored without finding

[3]To avoid the problem of multiple definition of variables, we use the DSA (Dynamic Single Assignment) form [2].

any solution, then $PROG_b$ is conform with its specification.

3.3 MCSs on a path

In this paper, we extend our BMC approach to *locate* which part of the program may be responsible for the error found during the BMC step. More precisely, let CE be a counter-example found during the BMC step. CE is the solution of CSP which contains the constraints from the precondition, the negation of the postcondition, and the constraints based on the assignments collected on the faulty path. Let $PATH$ denote this last set of constraints. Then the constraint system $C_{path} = CE \cup PRE \cup PATH \cup POST$ is ***UN**satisfiable* since CE is a counter-example that satisfies $\neg\ POST$. An MCS of C_{path} is a set of constraints which must be removed in order to make C_{path} satisfiable. By definition, such an MCS is a possible *error localization* on the faulty path. This first localization step assumes that the error is an assignment on the counter-example path. But the program can also be wrong because of a bad choice on a conditional node. LocFaults also computes bonded MCS on paths which are built by changing some conditions on the initial faulty path.

3.4 Algorithm scheme

The inputs of our algorithm are the CFG of the program, CE the counter-example, b_{cond}, a bound on the number of conditions which are diverted and b_{mcs} a bound on the number of MCS which are computed on each path. CE is a set of values of the input variables of the program. Roughly speaking, our algorithm traverses the CFG using CE to select one or the other branch of each conditional node, and collects the constraints associated with the assignments on the induced path. It changes zero, one or at most b_{cond} decisions on this path. At the end of a path, the set of constraints which have been collected is unsatisfiable, and at most b_{mcs} MCSs are computed on this CSP.

More precisely LocFaults proceeds as follows :

- It first propagates CE on the CFG until the end of the faulty path has been reached. Then it computes atmost b_{mcs} MCSs on the current CSP. This is a first localization on the counter-example path.
- Then LocFaults tries to divert one condition. When the first conditional node *cond* is reached, LocFaults takes the opposite decision as the one induced by CE, and continues to propagate CE to the last CFG node. If the CSP built from this diverted path is satisfiable, there are two kinds of *suspicious error set* :
  - the first one is the condition *cond* itself. Indeed, changing the decision for *cond* makes CE satisfies the postcondition $POST$,
  - another possible cause of the error is that a bad assignment before *cond* had produced a wrong decision. Thus LocFaults also computes atmost b_{mcs} MCSs on the CSP that contains the constraints collected on the path that reaches *cond*.

  This process is repeated on each conditional node of the counter-example path.
- A similar process is then applied for diverting k conditions for all $k \leq k_{max}$. To increase efficiency, the conditional nodes which correct the program are marked with the number of diversions which have been made before they had been reached. For a given step k, if changing the decision of a conditional node *cond* marked with value k' with $k' \leq k$ corrects the program, this correction is ignored. In other words, we only consider the first time where a conditional node corrects the program.

4. EXPERIMENTS

To evaluate the capabilities of our approach we experimented with two sets of Benchmarks : the well known TCAS suite from Siemens[25], and a set of variations of the `Tritype` program. We compared the results of LocFaults with the one of BugAssist.

TCAS is an aircraft collision avoidance system. The program contains 173 lines of C code with almost no arithmetic operations. The suite contains 41 faulty versions .

The `Tritype` program is a standard benchmark in test case generation and program verification since it contains numerous non-feasible paths because of complex conditional statements in the program. The program takes three positive integers as inputs (i, j, k) the triangle sides, and returns the value 2 if the inputs correspond to an isosceles triangle, the value 3 if they correspond to an equilateral triangle, the value 1 if they correspond to some other triangle, and the value 4 otherwise. The `Tritype` program is also a decision problem but we derived from the initial program two versions with more arithmetic operations. The first returns the product of the length of the sides, whereas the second one computes the square of the surface of the triangle by using Heron's formula.

The results of these experiments are detailed in the next subsections. All experiments were done on an Intel Core Core i7-3720QM at 2.6 GHz with 8 GB of memory running 64-bit Linux. LocFaults uses the IBM solvers CP OPTIMIZER and CPLEX [4].

4.1 TCAS suite

The results of the experiments are reported on Table 1. First column specifies the TCAS version number, the second one the number of errors in the program and the third column gives the number of the generated counter-examples. The two last columns provide the number of errors that have been found by LocFaults and BugAssist. Some of the versions are omitted because the errors correspond to array index out of bound and we still cannot handle this kind of overflow errors. We didn't report the computation times because there is no significant difference. The reported results for LocFaults have been obtained with at most one deviation; except for version `V41` where two deviations were required.

The size of the set of suspicious instructions identified by BugAssist is in general larger than the sum of the sizes of the sets of suspicious instructions generated by LocFaults but BugAssist identifies a bit more errors than LocFaults. More importantly, since LocFaults reports a set of MCS for each faulty path, the error localization process is much more easier than with the single set of suspicious errors reported by BugAssist.

In all, the performances of LocFaults and BugAssist

[4] `http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/`

Version	Nb_E	Nb_CE	LF	BA
V1	1	131	131	131
V2	2	67	67	67
V3	1	23	23	13
V4	1	20	4	20
V5	1	10	9	10
V6	1	12	11	12
V7	1	36	36	36
V8	1	1	1	1
V9	1	7	7	7
V10	2	14	12	14
V11	2	14	12	14
V12	1	70	45	48
V13	1	4	4	4
V14	1	50	50	50
V16	1	70	70	70
V17	1	35	35	35
V18	1	29	28	29
V19	1	19	18	19
V20	1	18	18	18
V21	1	16	16	16
V22	1	11	11	11
V23	1	41	41	41
V24	1	7	7	7
V25	1	3	2	3
V26	1	11	7	11
V27	1	10	9	10
V28	1	75	74	58
V29	1	18	17	14
V30	1	57	57	57
V34	1	77	77	77
V35	1	75	74	58
V36	1	122	120	122
V37	1	94	21	94
V39	1	3	2	3
V40	2	122	72	122
V41	1	20	16	20

Table 1: Results on TCAS

are very similar on this benchmark well adapted for a Boolean solver.

4.2 Variations on the `Tritype` program

The results of the experiments on the different variations of the `Tritype` program[5] are reported in Table 2. In that table, the numbers are the line numbers and the red numbers are the injected errors that have been found by the tools. For LocFaults, the underlined numbers are conditions, one line corresponds to a path through the CFG, and contains either a condition alone, or condition and the assignments before that condition which allow to change the branch. For example, for `TritypeV1` in column = 1 where one condition is diverted, LocFaults first locates the condition 26 as being erroneous. Then it locates condition 48 and locates the assignments 30 or 25 which can be responsible of the bad decision on 48.

Versions 1 to 5 of `Tritype` correspond to the standard `Tritype` program where we injected different kinds of errors.

- `TritypeV1` : the error was introduced in the last assignment statement of the program. LocFaults identified this error in the first step, without deviating any condition.
- `TritypeV2` : the error is in a nested condition, just before the last assignment. LocFaults finds the relevant suspicious statement after 4 deviations. BugAssist identifies also the relevant suspicious statement.
- `TritypeV3` : the error is an assignment and will entail a bad branching. Here again, LocFaults only finds it after 4 deviations but all suspicious set contains only one statement.
- `TritypeV4`: the error is in a condition, at the beginning of the program. LocFaults finds it very quickly. Even the first identified suspicious statement may be helpful : it is an assignment, just after the wrong condition.
- `TritypeV5` : there are two wrong conditions in this program. LocFaults needs to divert 3 conditions to find the two errors, while BugAssist only finds the first one.
- `TritypeV6` : is a variation that returns the perimeter of the triangle. LocFaults identified this error in the first step, without deviating any condition.

Versions 7 and 8 of `Tritype` are some variations of the original program that return *non linear* expressions. They have the same control structure as `Tritype`. `TritypeV7` computes the product of the three sides and `TritypeV8` computes the square of the area of the triangle. The specification of `TritypeV8` uses the Heron formula $\sqrt{s(s-i)(s-j)(s-k)}$ where $s = (i+j+k)/2$. To ensure that the returned value is an integer, we compute the square of the area and we assume as precondition that s is even. Moreover, the returned value varies according to the triangle type. For example, if the triangle is isosceles and $i == j$, the returned value is $s(s-i)(s-i)(s-k)$, and if the triangle is equilateral, the returned value is $(3 \times i^4)/16$.

Computations times are very short for all programs but `TritypV7` and `TritypV8`. Table 3 reports the times for these two programs. P stands for the pre-processing time[6] whereas the other rows contain the solving time. LocFaults is an order of magnitude faster than BugAssist on these two benchmarks. This clearly shows the benefit of using a constraint solver for programs containing non-trivial numerical statements. Indeed, these numerical constraints are much more difficult to handle by the SAT solver used in BugAssist than by the CSP solver used in LocFaults.

On all these benchmarks, the size of the set of suspicious instructions identified by BugAssist is similar to the sum of the sizes of the sets of suspicious instructions generated by LocFaults. But these benchmarks also show that the debugging process is much easier with the small set provided by LocFaults than with the global set of suspicious instructions computed by BugAssist. Our approach is a flow-based approach, that generates the sets of suspicious instructions in an incremental way. It finds errors along the path of the counter-example, and reports some explanations in an order that can help the user to find the bug. This is more appropriate for debugging than a global approach like BugAssist which computes a single set of suspicious instructions.

[5]The source code of these benchmarks can be found at: `http://users.polytech.unice.fr/~rueher/Benchs/LocF/`

[6]For LocFaults this is the time needed for the JDT Eclipse parser to build the AST from the Java program.

Program	Counter-example	Errors	LocFaults				BugAssist
			= 0	= 1	= 2	= 3	
TritypeV1	$\{i = 2, j = 3, k = 2\}$	54	{54}	{26} {48},{30},{25}	{29, 32} {53, 57},{30},{25}	/	{26, 27, 32, 33, 36, 48, 57, 68}
TritypeV2	$\{i = 2, j = 2, k = 4\}$	53	{54}	{21} {26} {35},{27},{25} {53},{27},{25}	{29, 57} {32, 44}	/	{21, 26, 27, 29, 30, 32, 33, 35, 36, 33, 35, 36, 53, 68}
TritypeV3	$\{i = 1, j = 2, k = 1\}$	31	{50}	{21} {26} {29} {36},{31},{25} {49},{31},{25}	{33, 45}	/	{21, 26, 27, 29, 31, 33, 34, 36, 37, 49, 68}
TritypeV4	$\{i = 2, j = 3, k = 3\}$	45	{46}	{45},{33},{25}	{26, 32}	{32, 35, 49} {32, 35, 53} {32, 35, 57}	{26, 27, 29, 30, 32, 33, 35, 45, 49, 68}
TritypeV5	$\{i = 2, j = 3, k = 3\}$	32,45	{40}	{26} {29}	{32, 45} {35, 49},{25} {35, 53},{25} {35, 57},{25}	/	{26, 27, 29, 30, 32, 33, 35, 49, 68}
TritypeV6	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{32},{27}	/	/	{28, 29, 31, 32, 35, 37, 65, 72}
TritypeV7	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{27},{32}	/	/	{72, 37, 53, 49, 29, 35, 32, 31, 28, 65, 34, 62}
TritypeV8	$\{i = 3, j = 4, k = 3\}$	61	{61}	{29} {35},{30},{25}	/	/	{19, 61, 79, 35, 27, 33, 30, 42, 29, 26, 71, 32, 48, 51, 54}

Table 2: Tritype benchmark

Programme	LocFaults					BugAssist	
	P	L				P	L
		$= 0$	≤ 1	≤ 2	≤ 3		
TritypeV7	$0,722s$	$0,051s$	$0,112s$	$0,119s$	$0,144s$	$0,140s$	$20,373s$
TritypeV8	$0,731s$	$0,08s$	$0,143s$	$0,156s$	$0,162s$	$0,216s$	$25,562s$

Table 3: Computation times for non linear programs

5. DISCUSSION

5.1 Related work

Various techniques to support error localisation have been proposed in the test and verification community.

The problem of error localization was first addressed in the test community where many systems have been developed. The most famous one is Tarantula [16, 15] that uses different metrics to rank suspicious statements detected while running a battery of tests. The critical point of this approach is that it requires an oracle for deciding whether the test result is correct or not. To avoid this problem, we consider here the Bounded-Model Checking (BMC) framework where we only require a postcondition or an assertion to check.

In this BMC context Bal et al [1] developed one of the first error localization system. Roughly speaking, they perform multiple calls to a model checker and compare the trace of generated counter-examples with a correct execution trace. Transitions that are not included in the correct trace are reported as a possible cause of the error. Their algorithm has been implemented in the of SLAM BMC framework that verifies temporal safety properties of C programs.

More recently, approaches based on the derivation of correct traces were introduced in EXPLAIN[14, 13]. EXPLAIN works as follows:

1. Calling the model-checker CBMC[7] to find an execution that violates the postcondition;
2. Using a pseudo-Boolean solver for searching the nearest correct execution;
3. Computing the difference between both traces.

Then, EXPLAIN produces a propositional formula S associated with program `P` but whose assignments do not violate the specification. Finally, EXPLAIN extends S with constraints defining an optimization problem the goal of which is to find a satisfying assignment that is as close as possible to the counter example; proximity is measured by a distance on the execution of `P`.

An approach similar to one of EXPLAIN was introduced in [24] but it is based on testing rather than on model checking: the authors use a series of correct and incorrect tests and a distance metrics to select a correct test from a given set of test data. This approach assumes that a full oracle is available.

In [11, 12], the authors also start from a counter-example, but they use the specification to derive a correct program for the same input data. Each identified statement can be used to correct errors. This approach ensures that the errors are inside the set of suspicious statements (assuming that the error is in the considered erroneous model). In other words, their approach identifies a super set of erroneous statements. To reduce the number of potential errors, the process is restarted for various counter-examples and the intersection of the sets of suspicious statements are calculated. However, this approach suffers from two major problems:

- The search space may be very large since every expression can be modified;
- It generates numerous spurious diagnoses since any change in an expression is possible (for example, chang-

[7] `http://www.cprover.org/cbmc/`

ing the last assignment of a function to return the expected result).

To overcome these problems, Zhang et al [28] proposed to change only predicates of the control flow. The intuition behind this approach is that by switching the results of a predicate and modifying the control flow, the program state cannot only be inexpensively modified, but in addition, it is often possible to reach a successful state. Liu et al [22] generalized this approach by editing multiple predicates. They also propose a theoretical study of debugging algorithm for *RHS* errors, that is, errors in predicate control and in the right hand side of assignments.

In [3], the authors address the problem of analyzing the trace of a counter-example and of error localization in the context of formal verification of hardware systems. They use the notion of causality introduced by Halpern and Pearl to formally define a set of causes of the violation of the specification by a counter-example.

Manu Jose and Rupak Majumdar [17, 18] have addressed this problem differently: they introduced a new algorithm that uses a MAX-SAT solver to calculate the maximum number of clauses of a Boolean formula that can be satisfied by an assignment. Their algorithm works in three steps:

1. They encode a trace of a program by a Boolean formula F that is satisfiable if and only if the trace is satisfiable;
2. They build a false formula F' by requiring that the postcondition is true (the formula F' is unsatisfiable because the trace models a counter-example that violates the postcondition);
3. They use MAXSAT to compute the maximum number of clauses that can be satisfied in F' and display the complement of this set as a potential cause of errors. In other words, they calculate the complement of a MSS (Maximum Satisfiable Subset).

Manu Jose and Rupak Majumdar [17, 18] have implemented their algorithm in BugAssist, a state-of-art verification tool based on CBMC.

Si-Mohamed Lamraoui and Shin have recently developed SNIPER, a tool that generalizes the approach of BugAssist in order to improve error localization in programs with multiples faults. SNIPER calculates the MSS of a formula $\psi = EI \wedge TF \wedge AS$ where EI encodes the erroneous input values, TF denotes a formula modelling all paths of the program, and AS is the violated assertion. The MCSs are obtained by taking the complement of the calculated MSS. The implementation is based on the intermediate representation LLVM and the SMT solver Yices. The authors compared SNIPER with BugAssist on the Siemens test suite TCAS. SNIPER identified by anywhere 5% more errors than BugAssist but required about much more times than BugAssist on this benchmark. More importantly, their approach assumes that a set of inputs that triggers all faults in the program is available.

In [27], the authors propose to compute irreducible infeasible subsets of constraints. They use a constraint solver to derive the MCS from these sets but the number of single fault candidates they generate is rather large.

There are some similarities between the approach we propose here and the framework introduced by Manu Jose and Rupak Majumdar. The main differences are:

1. We use the control flow graph to collect the constraints on the path from the counter example, as well as on the derived paths from the path of the counter-example by assuming that at most k conditional statements may contain errors. So, we do not transform the whole program into a system of constraints.
2. We use more general algorithms than MAXSAT that make it easier to deal with numerical constraints.

5.2 Contribution and future work

Our flow-based and incremental approach is a good way to help the programmer with bug hunting since it locates the errors around the path of the counter-example. Further work concerns programs with loops where scalability may be an issue. We plan also to develop an interactive version of our tool that provides the localizations one after the others, and takes benefit from the user knowledge to select the condition that must be diverted.

Furthermore, the constraint-based framework that we have introduced is well adapted for handling arithmetic operations. Moreover, it can be extended in straightforward way for error-localization in programs with floating-point numbers computations. Such an extension would be more difficult in SAT-based framework of BugAssist, or in SNIPER where the whole program is transformed in anLLVM intermediate representation.

6. REFERENCES

[1] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In Proceedings of POPL, pages 97–105. ACM, 2003.

[2] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In PASTE'05, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, pages 82–87. ACM, 2005.

[3] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler. Explaining counterexamples using causality. In Proceedings of CAV, volume 5643 of Lecture Notes in Computer Science, pages 94–108. Springer, 2009.

[4] E. Birnbaum and E. L. Lozinskii. Consistent subsets of inconsistent systems: structure and behaviour. J. Exp. Theor. Artif. Intell., 15(1):25–46, 2003.

[5] J. W. Chinneck. Localizing and diagnosing infeasibilities in networks. INFORMS Journal on Computing, 8(1):55–62, 1996.

[6] J. W. Chinneck. Fast heuristics for the maximum feasible subsystem problem. INFORMS Journal on Computing, 13(3):210–223, 2001.

[7] J. W. Chinneck. Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods. Springer, 2008.

[8] H. Collavizza, M. Rueher, and P. V. Hentenryck. Cpbpv: a constraint-programming framework for bounded program verification. Constraints, 15(2):238–264, 2010.

[9] H. Collavizza, N. L. Vinh, O. Ponsini, M. Rueher, and A. Rollet. Constraint-based bmc: a backjumping strategy. STTT, 16(1):103–121, 2014.

[10] A. Felfernig, M. Schubert, and C. Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. AI EDAM, 26(1):53–62, 2012.

[11] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to c. In Proceedings of CAV, volume 4144 of Lecture Notes in Computer Science, pages 358–371. Springer, 2006.

[12] A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for c programs. Electr. Notes Theor. Comput. Sci., 174(4):95–111, 2007.

[13] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. STTT, 8(3):229–247, 2006.

[14] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In Proceedings of CAV, volume 3114 of Lecture Notes in Computer Science, pages 453–456. Springer, 2004.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In ASE, IEEE/ACM International Conference on Automated Software Engineering, pages 273–282. ACM, 2005.

[16] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In ICSE,Proceedings of the 22rd International Conference on Software Engineering, pages 467–477. ACM, 2002.

[17] M. Jose and R. Majumdar. Bug-assist: Assisting fault localization in ansi-c programs. In Proceedings of CAV, volume 6806 of Lecture Notes in Computer Science, pages 504–509. Springer, 2011.

[18] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In Proceedings of PLDI, pages 437–446. ACM, 2011.

[19] U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In Proceedings of AAAI, pages 167–172. AAAI Press / The MIT Press, 2004.

[20] M. H. Liffiton and A. Malik. Enumerating infeasibility: Finding multiple muses quickly. In Proc. of CPAIOR, volume 7874 of Lecture Notes in Computer Science, pages 160–175. Springer, 2013.

[21] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning, 40(1):1–33, 2008.

[22] Y. Liu and B. Li. Automated program debugging via multiple predicate switching. In Proceedings of AAAI. AAAI Press, 2010.

[23] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On computing minimal correction subsets. In Proc. of IJCAI. IJCAI/AAAI, 2013.

[24] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In Proceedings of ASE, pages 30–39. IEEE Computer Society, 2003.

[25] D. S. Rosenblum and E. J. Weyuker. Lessons learned from a regression testing case study. Empirical Software Engineering, 2(2):188–191, 1997.

[26] M. Tamiz, S. J. Mardle, and D. F. Jones. Detecting iis in infeasible linear programmes using techniques from goal programming. Computers & OR, 23(2):113–119, 1996.

[27] F. Wotawa, M. Nica, and I. Moraru. Automated debugging based on a constraint model of the program and a test case. J. Log. Algebr. Program., 81(4):390–407, 2012.

[28] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In Proceedings of ICSE, pages 272–281. ACM, 2006.